(19) JAPANESE PATENT OFFICE

JPA 64-046840

PATENT ABSTRACTS OF JAPAN

(11) Publication number: **01046840 A**

(43) Date of publication of application: **21.02.89**

(51) Int. Cl **G06F 9/44**

(21) Application number: **62203712**

(22) Date of filing: **17.08.87**

(71) Applicant: **TOSHIBA CORP**

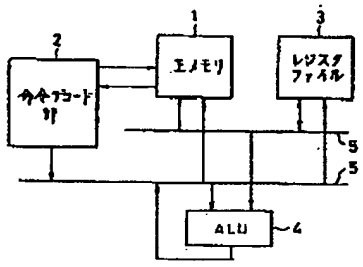
(72) Inventor: **KOYAMA KIYOMI**

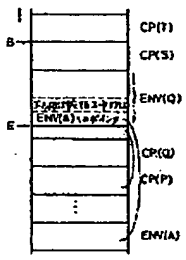
(54) PROLOG PROCESSOR

(57) Abstract:

PURPOSE: To increase the prolog processing speed by deciding the goal of a main body part when a prolog language is compiled and omitting the relevant processing when no allocation of a back track point is required to a local stack.

CONSTITUTION: The titled processor is composed of a main memory 1, an instruction decoding part 2, a register file 3 and an arithmetic and logic unit ALU4. Then it is checked whether a selection point frame CP is produced or not when a prolog language is compiled and the goal of a main body part is carried out. If not, a process where a back track point is allocated to a local stack can be omitted. Thus the memory access for said allocation is eliminated and the prolog processing speed is increased.





⑲日本国特許庁(JP)　⑪特許出願公開

⑫公開特許公報(A)　昭64－46840

⑤Int.Cl.⁴	識別記号	庁内整理番号	㊸公開　昭和64年(1989)2月21日
G 06 F 9/44	330	B－8724－5B	

審査請求　未請求　発明の数 1　(全13頁)

㊹発明の名称　PROLOG処理装置

㉑特　願　昭62－203712
㉒出　願　昭62(1987)8月17日

㉒発明者　小山　清美　神奈川県川崎市幸区小向東芝町1番地　株式会社東芝総合研究所内
㉑出願人　株式会社東芝　神奈川県川崎市幸区堀川町72番地
㉔代理人　弁理士　鈴江　武彦　外2名

明　細　書

1．発明の名称

PROLOG処理装置

2．特許請求の範囲

(1)PROLOGプログラムを解釈し、同一のローカルスタック上に環境フレーム、コンティニュエーションフレーム及び選択点フレームを形成しながらプログラムの実行を進めるための上記プログラムに対応したマシン命令を生成するコンパイラと、このコンパイラによって生成された前記命令コードを記憶するとともに前記ローカルスタックを提供するメモリと、このメモリに記憶されたマシン命令を実行する命令実行部とを備えたPROLOG処理装置において、前記コンパイラは、サブゴールを起動する必要のないゴールをリストアップしたテーブルと、前記プログラム中にカットオペレータを含む節が存在するとき、その節の実行中に新たに選択点フレームを生成する可能性の有無を前記テーブルに基づき判定する第1の判定手段と、この第1の判定手段により新たな選択点フレームが生成されると判定された場合には、その節に続く同一の頭部を持つ他の節が存在するかどうかを判定する第2の判定手段と、この第2の判定手段により他の節が存在すると判定された場合には前記ローカルスタックのトップに現在の選択点フレームよりも一つ手前の前選択点フレームへのポインタをバックトラックポイントとして格納する第1のマシン命令を生成し当該節の環境フレームを形成するマシン命令の前に挿入する第1のマシン命令生成手段と、前記第2の判定手段によって前記他の節が存在しないと判定された場合には前記ローカルスタックのトップに現在の選択点フレームへのポインタをバックトラックポイントとして格納する第2のマシン命令を生成し当該節の環境フレームを形成するマシン命令の前に挿入する第2のマシン命令生成手段と、上記第1及び第2のマシン命令生成手段によるマシン命令の生成が行われた場合に現環境フレームへのポインタから所定のオフセット量を加えたローカルスタック上の位置からデータを読出して現選択点フ

レームのポインタ用のレジスタにロードする第3のマシン命令を生成しカットオペレータを実行するマシン命令の位置に挿入する第3のマシン命令生成手段と、前記第1の判定手段によって新たな選択点フレームが生成されないと判定された場合には、その節に続く同一の頭部を持つ他の節が存在するかどうかを判定する第3の判定手段と、この第3の判定手段によって他の節が存在すると判定されたときに、前選択点フレームへのポインタを前記ローカルスタックから読み出して現選択点フレームのポインタ用のレジスタにロードする第4のマシン命令を生成しカットオペレータを実行するマシン命令の位置に挿入する第4のマシン命令生成手段とを具備したものであることを特徴とするＰＲＯＬＯＧ処理装置。

(2)前記コンパイラは、前記カットオペレータを含まない節に対しても、その節の実行中に新たな選択点フレームを生成する可能性の有無を前記テーブルに基づいて判定し、新たな選択点フレームが生成されないと判定された場合には、前記環境フレーム又は前記コンティニュエーションフレームの生成を行なわないマシン命令を生成することを特徴とする特許請求の範囲第1項記載のＰＲＯＬＯＧ処理装置。

3．発明の詳細な説明

〔発明の目的〕

（産業上の利用分野）

本発明は論理型言語であるＰＲＯＬＯＧで記述されたプログラムを実行するＰＲＯＬＯＧ処理装置に関するものである。

（従来の技術）

ＰＲＯＬＯＧ言語で書かれたプログラムは一般に

Ａ：－Ｂ，Ｃ，Ｄ　　…①

の形をとる。：－記号の左辺にあるＡを頭部（head）、：－記号の右辺を本体（body）と呼ぶ。本体に含まれるＢ，Ｃ，Ｄをゴール（goal）と呼び、また全体を節（clause）と呼ぶ。上記①の節の意味するところは、「Ａが真であるためには、Ｂ，Ｃ，Ｄが真でなければならない」、換言すればＢ，Ｃ，Ｄが全て成立すればＡは真である」である。

プログラムの実行過程で、この節が後述する一時変数（temporary variable）のみ持つ場合頭部及び本体のゴール間での引数の値の受渡しはレジスタを通して行われ、この節が後述する永久変数（permanent variable）を持つ場合には、ローカルスタックに変数の値セル（value cell）が形成されここを通して変数の値の引渡しが行われる。この引渡しの為にローカルスタックに形成されるフレームを環境（environment）フレーム（ＥＮＶ）と呼び、文献Ｄ．Ｈ．Ｄ．Warren,"An Abstract ＰＲＯＬＯＧ Instruction Set", ＡＩCenter,ＳＲＩ International, August １９８３（以下文献１と呼ぶ）によれば第９図に示すような構成である。即ち、まず初めにこのＥＮＶの直前にローカルスタック上に割付けられた前ＥＮＶへのポインタ（継続環境）が形成され、次いで本節の実行が成功に終わった後で次に実行すべき親ゴールに対応するコードアドレス（継続コード）が形成され、その後、本節の永久変数の値セルが形成される。また当該節と同じ頭部を持つ他の節がプロセジャ中に含まれ、このプロセジャが親ゴールから呼ばれた場合、親ゴールから渡される引数や、プロセジャ実行直前のプロセッサ状態を格納するためにローカルスタック上にフレームが形成される。これを選択点（choice point）フレーム（ＣＰ）と呼び文献１によれば第１０図に示すような構成となる。このフレームにはＰＲＯＬＯＧの実行において必要となる他のスタックへのポインタ、例えばヒープポインタ（Ｈ）、トレイルポインタ（ＴＲ）などの外、このＣＰの直前にローカルスタック上に生成されたＣＰへのポインタ（Ｂ）、ある節の実行が失敗（fail）してバックトラックが発生した時に、代わりに実行すべき節に対応するコードアドレスなどが格納され、更に親ゴールから渡される引数（ゴール引数１～ゴール引数Ｍ）が格納される。

ところである節が永久変数を持つか否かの判定法は、文献１に示されている。これによれば、当

該節が本体部に２つ以上のゴールを持ち、ある変数が頭部および本体部の第１ゴールのうちの少なくとも一方に現われた場合を１回とカウントし、以後現われるその変数を２回からカウントすると、２回以上現われた変数を永久変数と呼ぶ。永久変数以外の変数が一時変数である。永久変数を持つ節では、既に述べたようにローカルスタックにＥＮＶが割付けられる。更に文献１の規準によれば、ある節の本体部のゴール数が２つ以上であって当該節が永久変数を持たない場合、コンティニューエーション（ＣＯＮＴ）と呼ばれるフレームをローカルスタックに割付ける。ＣＯＮＴはＥＮＶから永久変数の値セルを除いた部分、即ち第９図の継続環境のスロットと継続コードのスロットとで構成され、ＥＮＶと同様に現ＥＮＶポインタ（Ｅレジスタ）でローカルスタック上での位置が示される。

先にあげた①の節が永久変数を持つと仮定すると、この節に対応するコンパイルコードは第１１図のようになる。本図で１行目のallocate命令はローカルスタックにＥＮＶを割付ける機能を持ち、８行目のdeallocate命令は逆にローカルスタックからＥＮＶを取除く命令である。更に４行目および６行目のcall命令はリターンポインタをセットした上で、オペランドのプロセジャに実行を移す命令である。また９行目のexecute 命令では、call命令と同様にオペランドのプロセジャに実行を移すが、リターンポインタのセットは行わず、直前のdeallocate命令実行時にＥＮＶからリストアした継続コードをリターンポインタとして使う。２行目の get－args－of命令および３．５．７行目の put－args－of命令は擬似マシン命令である。get－args－of命令は親ゴールからアーギュメントレジスタを介して渡された引数と節の引数とのマッチングをとるget 命令の総称で、また put－args－of命令は本体部のゴールの引数をアーギュメントレジスタにロードするput 命令の総称である。各マシン命令の詳細動作は文献１に詳しいので、ここではこれ以上の説明を省略する。第１１図から明らかな様にＥＮＶは永久変数を含む節を実行する直前にローカルスタックに割付けられ、また本体部の最終ゴール実行直前にローカルスタックから取除かれる。

次にＰＲＯＬＯＧのプログラム実行中にＥＮＶやＣＰがローカルスタックに生成、棄却される様子を示す。例として第１２図（ａ）のプログラムを使う。この図で節１が最上位のゴールであり、本体部にＰ，Ｑ，Ｒのゴールを含んでいる。節２，３はゴールＰに対応するプロセジャを構成し、節４，５はゴールＱに対応するプロセジャを構成する。また節４のゴールＳに対応して節６，７，８がプロセジャを構成し、ゴールＴに対応して節９，１０がプロセジャを構成する。ここで、節１，２，４，６，９を実行する場合を考える。節４は永久変数を持たず、節９が永久変数を持つと仮定する。この時のローカルスタックの動きを第１２図（ｂ）に示す。図でＥＮＶ（Ａ）はゴールＡのＥＮＶ，ＣＰ（Ｐ）はプロセジャＰのＣＰ、ＣＯＮＴ（Ｑ）はゴールＱのＣＯＮＴを示している。

先ず節１の実行直前にＥＮＶ（Ａ）が、ローカルスタックに形成され、次いで節２でＰのプロセジャが別解を持つためＣＰ（Ｑ）がローカルスタックに形成され、次に節４で同じくＱのプロセジャが別解を持つためＣＰ（Ｑ）が、ローカルスタックに形成される。節４では本体部に３個のゴールがありしかも永久変数を持たないため、ローカルスタックにはＣＰ（Ｑ）に次いでＣＯＮＴ（Ｑ）が形成される。節４のゴールＳ，Ｔを実行する過程でプロセジャＳとＴに対応して各々ＣＰ（Ｓ），ＣＰ（Ｔ）が形成される。節９が永久変数を持つために更にＥＮＶ（Ｔ）が形成される。図に示されるようにＥＮＶ（Ｔ）の継続コードの位置には節４のゴールＵに対応するコードアドレスが格納され、またＥＮＶ（Ｔ）の継続環境の位置にはＣＯＮＴ（Ｑ）へのポインタが格納される。従って節９の実行でゴールＶ、ゴールＷと成功して、次にゴールＸの実行が失敗（fail）してバックトラックが起こると、ＣＰ（Ｔ）に格納された代替節のアドレス、即ち節１０に実行が移るが、ゴールＸの実行が成立すると、既に第１１図で述べた様

にゴールXの実行に先立ち、ＥＮＶ（Ｔ）の接続コード、継続環境の情報がレジスタにリストアされＥＮＶ（Ｔ）がＣＯＮＴ（Ｑ）まで巻戻されているため、節４のゴールＵに実行が移ることになる。節４に対応するＣＯＮＴ（Ｑ）にしても以上と同様で、ゴールＵの実行が成功に終わると、ＣＯＮＴ（Ｑ）からＥＮＶ（Ａ）まで巻戻され、節１のゴールＲに実行が移る、という形で節１全体の実行が推移する。

この様にＰＲＯＬＯＧプログラムを実行する過程でローカルスタックにＣＰやＥＮＶ．ＣＯＮＴのフレームを頻繁に形成することが必要となる。

更に次には本体部にカットオペレータが含まれる場合、ローカルスタックにこれらのフレームがどう形成され、棄却されるかを示す。例として次の２つの節からなるＰＲＯＬＯＧプログラムを考える。

Ａ：－Ｂ，！，Ｃ

）②

Ａ：－Ｄ

このプログラムの第１節に本体部でゴールＢとゴールＣの間にカットオペレータ！が含まれている。上記②のプログラムの意味するところは、ゴールＢの実行が成功した場合には、ゴールＣの実行に移るが、この場合他の同じ頭部を持つ節の実行を放棄する。従って、もしゴールＣが成功すれば節１全体の実行が成立するが、仮りにゴールＣの実行が失敗に終わっても節２の実行は試されない。即ち節２の実行が試されるのは節１でカットオペレータを左から右へ通過する以前、言い換えれば節１のゴールＢが不成功に終わる場合のみである。このことからプログラム②はカットオペレータを使わずに次の様に書くこともできる。

Ａ：－Ｂ，Ｃ

）③

Ａ：－not（Ｂ），Ｄ

しかし乍ら、上記２つのプログラム②，③は一般的に同一のセマンティックスで動作すると考えることができるものの、文献Ｗ．Ｆ．Clocksin, et al “Programming in Prolog” Springer－Verlag（以下文献２と呼ぶ）に示されるように、実行速度に於いて両者は大きく食い違う可能性を持っている。即ち、ゴールＢが非常に複雑で実行に多くの時間を費やすとした場合、プログラム②ではゴールＢの成功／不成功を１回だけ確かめれば済むのに対し、プログラム③では、２回必要となる。プログラム③の方が実行により多くの時間を費やし、カットオペレータを使ったプログラム②の方が実行効率が良い。

また、カットオペータはローカルスタック（メモリ）の使用効率を向上させる効果もある。これを第１３図（ａ），（ｂ）を使って説明する。第１３図（ａ）のプログラムにおいて、節１を実行する場合を考える。節１のゴールＰを実行するため先ず節２を実行すると、この節には別解があるため第１３図（ｂ）のＣＰ（Ｐ）がローカルスタックに生成される。節１のゴールＱを実行する場合も、節４を実行すると、同様に第１３図（ｂ）のＣＰ（Ｑ）が生成される。ここで節４が永久変数を持つとすると、第１３図（ｂ）のＥＮＶ（Ｑ）が生成される。本体部のゴールＲ，Ｓを実行するため、節６、節８を実行すると、共に別解を持つため、第１３図（ｂ）のようにＣＰ（Ｒ），ＣＰ（Ｓ）がローカルスタックに生成される。さて節４でゴールＳの成功の後カットオペレータを左から右へ通過すると、この時点で仮りにその後ゴールＴの実行が不成功に終わったとしても、バックトラックしてＳの別解を求めるために節９を試したり、またＲの別解を求めるために節７を試したりすることが不要となる。また既に述べたようにゴールＴの実行が不成功に終わった場合、Ｑの別解を求めるために節５が試されることもない。以上の事情から、節４の本体部でカットオペレータを左から右へ通過した時点で第１３図（ｂ）のローカルスタックでＣＰ（Ｓ），ＣＰ（Ｒ）が不要となり、更にゴールＴが不成功の場合、第１３図（ｂ）のＥＮＶ（Ｑ），ＣＰ（Ｑ）も不要となる。この様にカットオペレータによってそれ以後の実行で不要なフレームが明確になり、これをローカルスタックから順次取除くことによって効率良く

ローカルスタックが使用されることになる。

以上述べた様にカットオペレータを使うことによってプログラムの実行が高速化され、またローカルスタックの使用効率が向上するという利点が得られるが、ＰＲＯＬＯＧプログラムを実行するマシン上でカットオペレータをどう実現するかが大きな課題となる。即ち、通常ＣＰやＥＮＶなどローカルスタック上に形成されるフレームは第１１図に示したような明確なマシン命令によって生成，廃棄が行なわれるが、カットオペレータについてはそれに対応するマシン命令は明確に規定されていない。しかも以上の説明でも明らかな様に、カットオペレータを使用した場合のロールスタックからフレームを解放する情況は、あるＣＰから直前のＣＰに巻戻す、或いはあるＥＮＶから直前のＥＮＶに巻戻すという単純な規則には従わず、不特定多数個のフレームを飛び越して前に戻るということが発生するため、巻戻す位置をどう効率よく見つけるか、どの時点で巻き戻せば副作用を発生させずに済むかといった点でカットオペレータが高速で処理できるインプリメンテーションを考えることは非常に重要な課題になっている。

（発明が解決しようとする問題点）

このように、従来、ＰＲＯＬＯＧプログラムを高速で実行したり、メモリの使用効率を高める上でカットオペレータが重要な役割を担っているにも拘らず、実際にカットオペレータをＰＲＯＬＯＧマシンにインプリメントして効率良く処理することが困難であるという問題があった。

本発明は、このような事情に鑑みてなされたもので、カットオペレータを効率良く処理してＰＲＯＬＯＧプログラムの高速実行を可能とするＰＲＯＬＯＧ処理装置を提供することを目的とする。

更に本発明はカットオペレータを含まない場合も、ローカルスタックへのフレーム生成や廃棄といったコストの高い処理を最少限にとどめて高速実行が可能なＰＲＯＬＯＧ処理装置を提供することを目的とする。

［発明の構成］

（問題点を解決するための手段）

本発明は、ＰＲＯＬＯＧプログラムを解釈し、同一のローカルスタック上に環境フレーム（ＥＮＶ）、コンティニュエーションフレーム（ＣＯＮＴ）及び選択点フレーム（ＣＰ）を形成しながらプログラムの実行を進めるための上記プログラムに対応したマシン命令を生成するコンパイラと、このコンパイラによって生成された前記命令コードを記憶するとともに前記ローカルスタックを提供するメモリと、このメモリに記憶されたマシン命令を実行する命令実行部とを備えたＰＲＯＬＯＧ処理装置において、前記コンパイラを次のように構成したことを特徴としている。

即ち、前記コンパイラは、サブゴールを起動する必要のないゴールをリストアップしたテーブルと、前記プログラム中にカットオペレータを含む節が存在するとき、その節の実行中に新たにＣＰを生成する可能性の有無を前記テーブルに基づき判定する第１の判定手段と、この第１の判定手段により新たなＣＰが生成されると判定された場合には、その節に続く同一の頭部を持つ他の節が存在するかどうか（別解を持つかどうか）を判定する第２の判定手段と、この第２の判定手段により他の節が存在すると判定された場合には前記ローカルスタックのトップ（ＴＯＳ）に現在のＣＰよりも一つ手前の前ＣＰへのポインタをバックトラックポイントとして格納する第１のマシン命令を生成し当該節のＥＮＶを形成するマシン命令の前に挿入する第１のマシン命令生成手段と、前記第２の判定手段によって前記他の節が存在しないと判定された場合には前記ＴＯＳに現在のＣＰへのポインタをバックトラックポイントとして格納する第２のマシン命令を生成し当該節のＥＮＶを形成するマシン命令の前に挿入する第２のマシン命令生成手段と、上記第１及び第２のマシン命令生成手段によるマシン命令の生成が行われた場合にＥＮＶへのポインタから所定のオフセット量を加えたローカルスタック上の位置からデータを読出して現ＣＰへのポインタ用のレジスタ（以下Ｂレジスタと呼ぶ）にロードする第３のマシン命令を生成しカットオペレータを実行するマシン命令の

位置に挿入する第３のマシン命令生成手段と、前記第１の判定手段によって新たなＣＰが生成されないと判定された場合には、その節に続く同一の頭部を持つ他の節が存在するかどうか（別解を持つかどうか）を判定する第３の判定手段と、この第３の判定手段によって他の節が存在すると判定されたときに、前ＣＰへのポインタを前記ローカルスタックから読み出してＢレジスタにロードする第４のマシン命令を生成しカットオペレータを実行するマシン命令の位置に挿入する第４のマシン命令生成手段とを具備したものである。

（作　用）

カットオペレータを含む節で、実行中新たにＣＰを生成しない場合には、カットオペレータを左から右へ通過後、バックトラックが起きた場合のバックトラックポイントは、ＣＰのポインタ内容が当該節の実行開始時と変わらないため、現在のＣＰポインタを使って得ることができる。即ち、当該節に対応するプロセジャにＣＰがある場合（別解がある場合）はＢレジスタで指示されるＣＰに含まれる前ＣＰへのポインタをＢレジスタに戻せばよいし、またＣＰが無い場合（別解が無い場合）は現在のＢレジスタの値がそのままバックトラックポイントになる。従って、何れの場合も新たにローカルスタックにバックトラックポイントを格納する必要が無いし、通常のバックトラックと同じＢレジスタをターゲットにしてカットオペレータのバックトラックが実行できる。

逆にカットオペレータを含む節で、実行中新たにＣＰを生成する可能性がある場合は、当該節のゴールを実行する過程でバックトラックポイントが他のＥＮＶやＣＰに埋もれてしまう。このため当該節を実行開始直後にローカルスタックにバックトラックポイントを格納しておく。即ち、当該節に対応するプロセジャにＣＰがある場合は前ＣＰへのポインタを、またＣＰが無い場合は現ＣＰへのポインタを格納しておき、カットオペレータを左から右へ実行する時に、Ｂレジスタにバックトラックポイントを戻しておく。既に述べたように、ＥＮＶポインタからのオフセットで、このバックトラックポイントがアクセスできるように、文献１で述べられた規則とは別に本体部のゴールが１の時もＣＯＮＴをローカルスタックに割付けておく。このようにして、カットオペレータの右側のゴールで失敗してカットオペレータのバックトラックが起きた場合、通常のバックトラック処理でカットオペレータが処理できる。

更に本体部にカットオペレータを含むか否かに拘らず、本体部のカットオペレータ以外のゴールが、実行中にサブゴールの起動を行わず実行できる述語である場合、文献１の規準とは拘りなく、ローカルスタックにＥＮＶやＣＯＮＴのフレームを生成しないで実行する。

本発明によればコンパイル時に本体部のゴール実行中にＣＰが生成されるか否かチェックし、されない場合、ローカルスタックにバックトラックポイントを割付ける処理が省けるので、この割付けのためのメモリアクセスが減らせ、スピードアップの効果が得られる。また、従来の規則によらない新しい規則でローカルスタックへのＣＯＮＴフレームの割付けを行なうことで、バックトラックポイントをローカルスタックに格納した場合、必ず現ＥＮＶポインタのオフセットで取出せる利点がある。しかも、僅か４種の命令をＥＮＶ形成命令の前か、カットオペレータの位置に生成すれば良く、コンパイルが簡単になる。また本体部実行中に新たにＣＰを生成するか否かをコンパイル時にチェックして、不必要なフレームの生成を極力減らしており、この面でも大幅なスピードアップ効果が期待できる。

（実施例）

以下、本発明を図示の実施例に基づいて詳細に説明する。

第１図は、本実施例に係るＰＲＯＬＯＧ処理装置の構成を示す図である。

この装置は、主メモリ１、命令デコード部２、レジスタファイル３及び算術演算ユニット（ＡＬＵ）４で構成されている。主メモリ１は、ＰＲＯＬＯＧプログラムを格納する領域、上記ＰＲＯＬＯＧプログラムからマシン命令を生成するための

コンパイラを格納する領域、このコンパイラによって生成されたマシン命令を格納する領域、ローカルスタック領域、ヒープ領域、トレイル領域等を提供するメモリである。命令デコード部２は、上記主メモリ１に格納されたマシン命令を解釈する。このマシン命令に従って主メモリ１及びレジスタファイル３に格納されたデータがデータバス５を介してＡＬＵ４に送られ、適宜演算処理を施されてレジスタファイル３や主メモリ１に格納される。

上記コンパイラは、与えられたＰＲＯＬＯＧプログラムに対し、これに対応するマシン命令を形成する。もし、与えられたＰＲＯＬＯＧプログラムにカットオペレータ（！）を含む節が含まれている場合には、その節については、第２図に示すような処理が行われる。

先ずマシン命令として次の４つの機能を持ったものを追加する。第１はローカルスタックのトップオブスタック（ＴＯＳ）に現在のＣＰより１つ手前（ローカルスタックに割付けられた順番で）のＣＰへのポインタを格納してローカルスタックポインタの値を１増す命令（第１の命令）で、これを仮りにmark－prv －cp命令と呼ぶ。第２はローカルスタックのＴＯＳに現在のＣＰへのポインタ（即ちＢレジスタの内容）を格納してローカルスタックポインタの値を１増す命令（第２の命令）で、これを仮りにmark－cur －cp命令と呼ぶ。第３は現在のＥＮＶへのポインタ（即ちＥレジスタの内容）位置からあるオフセット（仮りに－１）を加えた位置から読み出して、Ｂレジスタにロードする命令（第３の命令）で、これを仮りにcut命令と呼ぶ。第４は現在のＣＰより１つ手前のＣＰへのポインタをローカルスタックから格納してＢレジスタにロードする命令（第４の命令）で、これを仮りにdcut命令と呼ぶ。これら４つの命令を使ってコンパイルコードを次の手順で生成して行く。

ＰＲＯＬＯＧの述語の中で、数値演算、論理演算、比較演算その他の組込み述語は、サブゴールを起動することなく実行できる。これらの述語を予め分類しておき、コンパイル時にコンパイラが参照できるような形、例えばテーブルにしておく。この分類分けされた述語をもとに、コンパイラは、カットオペレータを含む当該節が新たにＣＰを生成するか判定し（Ｓ１）、ＣＰを生成する可能性が無いと判定された場合は更に、この節を含むプロセジャのＣＰがあるか否か判定する（Ｓ６）。ＣＰが無い場合は何もしないが、ＣＰが有る場合は、カットオペレータ位置にdcut命令を生成する（Ｓ７）。一方、ステップＳ１でＣＰを生成する可能性があると判定された場合は、次いでその節を含むプロセジャにＣＰがあるか否かを判定し（Ｓ２）、ＣＰが無い場合はallocate命令の前にmark-prv-cp 命令を生成し（Ｓ３）、ＣＰが有る場合はallocate命令の前にmark-cur-cp 命令を生成する（Ｓ５）。両者の場合について、カットオペレータ位置にはcut 命令を生成する（Ｓ４）。

次に具体例に則して本実施例の装置の作用を説明する。

先ずカットオペレータを含む節が本体部実行中に新たにＣＰを生成する可能性があり、且つ、その節を含むプロセジャがＣＰを生成している場合、即ち第２図のＳ３の場合について第３図に基づいて説明する。第３図（ａ）はプログラムの一例である。今節１を実行するため本体部のゴールＰに対して節２を、ゴールＱに対して節４を実行して、かつ節４の本体部のゴールＳに対して節６を、ゴールＴに対して節８を実行するとする。節４はカットオペレータを含みしかも永久変数を持つと仮定すると、コンパイルコードは第３図（ｂ）の(1)～（11）の如くなる。(1)のtry －me－else命令でＱに対するＣＰがローカルスタックに形成され、その後(2)のmark－prv －cp命令で前ＣＰへのポインタがローカルスタックに格納される。そして(3)のallocate命令でＥＮＶが割付けられる。(5)，(6)の命令でゴールＳが実行されると、同図（ａ）の節６が実行され、また(7)，(8)の命令でゴールＴが実行されると、同図（ａ）の節８が実行される。節６，節８とも別解を持つためローカルスタックにＣＰが形成される。節８の実行時点でローカル

スタックは第３図（ｃ）の如くなる。本図から明らかなようにＥＮＶは同図（ａ）の節１と、節４に対応するもの（それぞれＥＮＶ（Ａ），ＥＮＶ（Ｑ））が割付けられており、ＥＮＶ（Ｑ）の継続環境としてＥＮＶ（Ａ）へのポインタが入っている。また現在の（最新の）ＣＰを指すポインタ・Ｂレジスタは ＣＰ（Ｔ）を、現在のＥＮＶを指すポインタ・Ｅレジスタは ＥＮＶ（Ｑ）を指している。

ＥＮＶ（Ｑ）の手前には同図（ｂ）の(2) mark-prv -cp命令で格納したバックトラックポイントがあり、ＣＰ（Ｐ）を指している。またＥＮＶ（Ｑ）には同図（ａ）の節４が永久変数を持つため、永久変数の値セルが格納されている。同図（ｂ）で実行を進めて(9)の cut 命令を実行する（これは同図（ａ）節４のカットオペレータを左から右へ通過することに相当する）とＥレジスタのオフセット、即ちこの場合（Ｅ－１）番地からバックトラックポイントを取出し、Ｂレジスタにロードする。この後、同図（ｂ）の（10）（12）の命令でゴールＴが実行されるが、（11）の deallocate命令でＥＮＶ（Ｑ）がローカルスタックから除かれ、その際、次に実行すべき命令として同図（ａ）のゴールＲに対応するコードアレドス２が設定されるので、ゴールＴの実行が成功するとゴールＲに実行が移る。またゴールＴの実行が失敗するとバックトラックが起こるが、既にＢレジスタにはバックトラックポインタとしてＣＰ（Ｐ）がセットされているため、ＣＰ（Ｐ）まで巻戻され、Ｐの代わりの節、同図（ａ）の節３が実行される。

以上はカットオペレータを含む節が永久変数を持つ場合であったが、次に永久変数を持たない場合を考える。同図（ａ）のプログラムで節４が永久変数を持たないと仮定する。この時もコンパイルコードは永久変数を持つ場合と同じで、同図（ｂ）のままである。但し、同図（ｂ）で３行目の allocate命令や１１行目の deallocate命令がＥＮＶを割付けるためではなく、ＣＯＮＴを割付ける目的を持つ。この結果ローカルスタックは第３図（ｄ）のようになる。即ち、同図（ｃ）でのＥＮＶ（Ｑ）が同図（ｄ）ではＣＯＮＴ（Ｑ）に変わっているのみである。ＥレジスタはやはりＣＯＮＴ（Ｑ）を指している。従って同図（ｂ）の（ａ）cut 命令では（Ｅ－１）番地からバックトラックポイントをＢレジスタにセットすることができる。

次にカットオペレータを含む節が本体部実行中に新たにＣＰを生成する可能性はあるが、その節を含むプロセジャがＣＰを生成していない場合、即ち第２図のＳ５の場合について第４図に基づき説明する。最初にカットオペレータを含む節が永久変数を持つ場合について説明する。第４図（ａ）に例としてあげたプログラムは、既に節を含むプロセジャがＣＰを生成している場合の説明で用いたものを一部変更して得たものである。即ち、カットオペレータを含む節５はプロセジャの最後に来ている。節１の実行に始まって、ゴールＰに対して節２、ゴールＱに対して節５を実行し、また節５の実行で、ゴールＳに対して節６、ゴールＴに対して節８を実行するものとする。節５のコンパイルコードは第４図（ｂ）の(1)～（12）の命令に置換される。(1)の trust -me-else fail 命令で、節４と節５に対してあったＣＰが廃棄される。この時点で節２のＣＰ、即ちＣＰ（Ｐ）は生成されている。(2)の mark-cur -cp命令では現ＣＰポインタの内容がローカルスタックのＴＯＳに格納される。この後(3)の allocate命令でＥＮＶが割付けられる。命令(5)，(6)で節６が、命令(7)，(8)で節８が実行される。節６，節８ともローカルスタックにＣＰを生成する。従ってこの時点でのローカルスタックは第４図（ｃ）の如くなる。現ＣＰポインタはＣＰ（Ｔ）を指し、現ＥＮＶポインタはＥＮＶ（Ｑ）を指している。そしてＥＮＶ（Ｑ）の手前にはＣＰ（Ｐ）を指すバックトラックポイントが格納されている。従って第４図（ｂ）で(9)の cut 命令を実行すると、（Ｅ－１）番地からバックトラックポイントが取出され、Ｂレジスタに格納される。（10），（12）の命令でゴールＵが実行されるが、この実行が成功すると、節１のゴ

ールRに実行が移り、この実行が失敗すると、バックトラックが起こり、ＣＰ（Ｐ）までＣＰが巻戻され、節２の代替の節である節３の実行が試される。

次にカットオペレータを含む節が永久変数を持たない場合、即ち第４図（ａ）の節５が永久変数を持たない場合について説明する。この場合のコンパイルコードも、永久変数を持つ場合と同じ、同図（ｂ）の如くなるが、節(3)のallocate命令、節（11）のdeallocate命令の機能が、ＥＮＶではなくＣＯＮＴをローカルスタックに生成，廃棄する処理に変わる。従って、この時のローカルスタックは同図（ｄ）のようになるがこの場合もバックトラックポイントはＥレジスタのオフセットで取出すことができる。

代わってカットオペレータを含む節の実行中、明らかにＣＰの生成が無いと判定された場合について述べる。先ずその節に対応するＣＰがローカルスタックにある場合、即ち第２図のＳ７の場合を第５図に基づき説明する。同図（ａ）はプログラム例である。節４は本体部にカットオペレータを含む。また変数Ｘは２つのゴールに現われるため、文献１の規則によれば永久変数であり、ＥＮＶの割付けが必要である。先ず文献１の規則通りにＥＮＶを割付けることを前提にして、説明を進める。第５図（ａ）で節１を実行するため、ゴールＰに対して節２を、ゴールＱに対して節４を実行するとする。節１でＥＮＶ、節２でＣＰ、節４でＣＰとＥＮＶが割付けられ、ローカルスタックは同図（ｃ）の如くなる。さて、問題の節４は同図（ｂ）の(1)～(9)の命令にコンパイルされる。(1)のtry－me－else命令でＣＰ（Ｑ）を生成し、(2)のallocate命令でＥＮＶ（Ｑ）を生成する。ここで注目すべきことは、allocate命令の前にmark－prv －cp命令やmark－cur －cp命令を置く必要はない。(4)，(5)の命令で var（Ｘ）ゴールを実行し、次のカットオペレータを(6)のdcut命令で処理する。これは、Ｂレジスタ即ちＣＰ（Ｑ）の前のＣＰ、即ちＣＰ（Ｐ）へのポインタをＢレジスタにロードする処理である。その後、ゴールＸ＝Ｙを(8)のunity Ｘ，Ｙ命令（文献１のunify 命令とは必ずしも厳密に対応していないことに注意）で実行する際、実行が成功した場合には節１のＲに実行が移り、実行が失敗した場合にはバックトラックガ発生して、Ｂレジスタが指すＣＰ（Ｐ）まで巻戻され、節Ｐが実行されることになる。

次に当該節に対応するＣＰが無い場合について第６図に基づき説明する。第６図（ａ）はそのプログラム例で、第５図（ａ）と比較して節４と節５の位置が入れ替わっているだけである。この場合のコンパイルコードは第６図（ｂ）の如くなる。(1)のtrust －me－elseは節４の実行時に生成したＣＰ（Ｑ）を廃棄し、従ってローカルスタックは第６図（ｃ）の如くなる。第５図（ｂ）の場合と違い、(4)，(5)の命令でvar （Ｘ）を実行した後、カットオペレータ位置では何もしない。即ち、次のＸ＝Ｙゴールを実行して失敗した後、カットオペレータを右から左へ通過しようとした時のバックトラックポイント、第６図（ｃ）のＣＰ（Ｐ）の位置がＢレジスタに残っているからである。従ってこの場合カットオペレータに対して何もする必要がない。既に述べたように節の本体部にあってサブゴールを起動しないで実行できる、従ってＣＰもＥＮＶも生成しないで実行できる述語は数値演算、論理演算、比較演算、その他の組込み述語だが、インプリメンテーションによっても異なるため、予めこれらを分類分けして、コンパイル時にコンパイラが参照できるテーブルなどにまとめておくことが必要である。

同様にこの方法に従えば、本体部にカットオペレータを含むか否かに拘らず、以上のように分類されたゴールのみで本体部が構成されている場合、その節の実行にあたっては、文献１の規則によれば永久変数を含みＥＮＶの割付けが必要な場合、或いはＣＯＮＴの割付けが必要な場合も、これらのフレームをローカルスタックに割付けることなく高速に実行できる。この場合、ゴール間の引数の受渡しはレジスタを通して行なう。この方法によれば、第５図（ａ）の節４は第７図の如く、又、第６図（ａ）の節５は第８図の如くコンパイルし

て実行することができる。

［発明の効果］

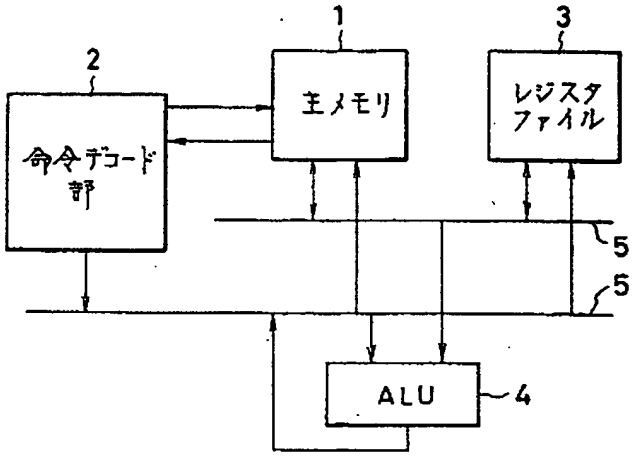
以上述べたように、本発明によれば、ＰＲＯＬＯＧ言語のコンパイル時に、本体部のゴールを判定して、ローカルスタックにバックトラックポイントを割付ける必要のないものは、その処理を省くようにしているので、この割付けのためのメモリアクセスを省略でき、処理のスピードアップを図ることができる。また、従来の規則によらない新しい規則でローカルスタックへのＣＯＮＴフレームの割付けを行なうことで、バックトラックポイントをローカルスタックに格納した場合、必ず現ＥＮＶポインタのオフセットで取出せるので、特にカットオペレータを含むＰＲＯＬＯＧプログラムを高速に処理できる。

４．図面の簡単な説明

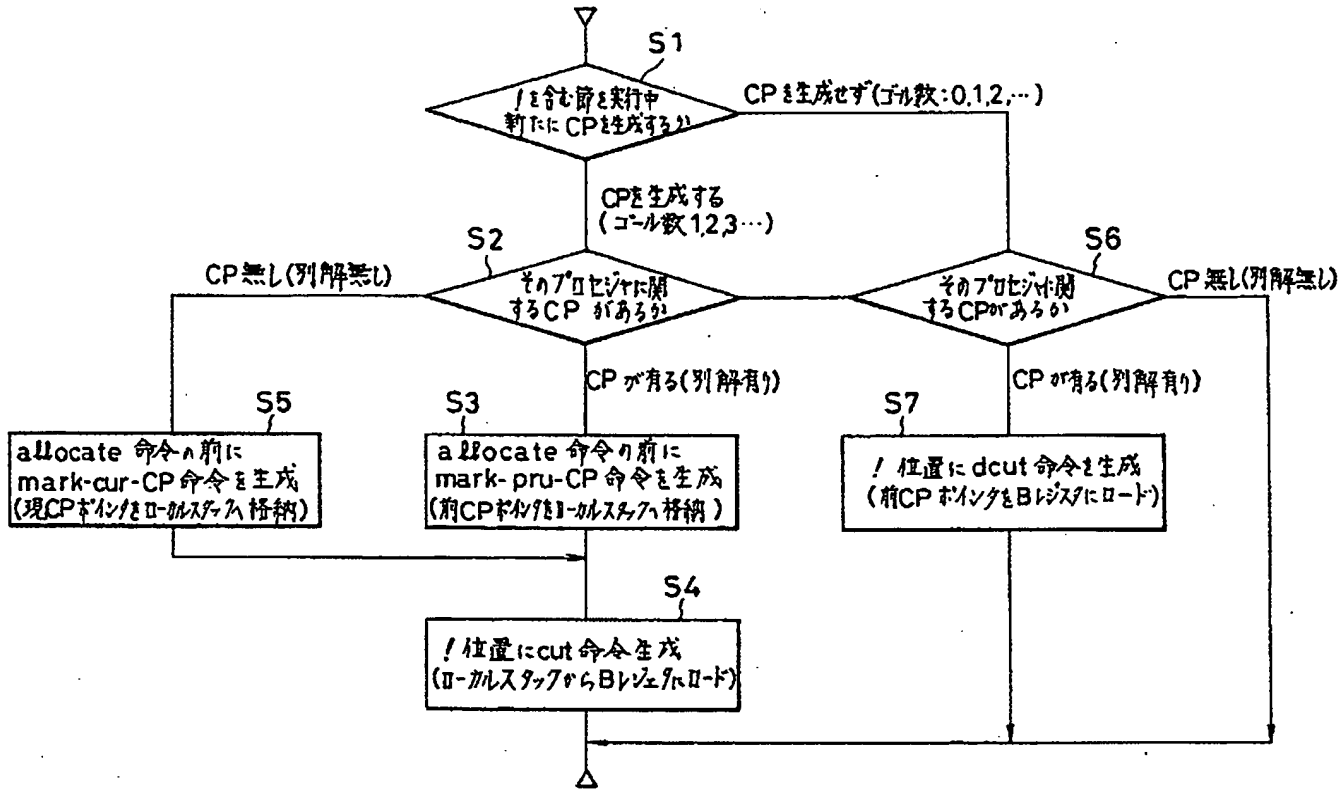
第１図～第８図は本発明の一実施例を説明するための図で、第１図は本発明の一実施例に係るＰＲＯＬＯＧ処理装置の構成を示すブロック図、第２図は同処理装置におけるコンパイラのカットオペレータの処理フローを示す流れ図、第３図は第２図のステップＳ３の処理例を示す図で、同図（ａ）はプログラム例、同図（ｂ）は同プログラムに対応するマシン命令、同図（ｃ）及び（ｄ）はローカルスタックの様子をそれぞれ示す図、第４図は第２図のステップＳ５の処理例を示す図で、同図（ａ）はプログラム例、同図（ｂ）は同プログラムに対応するマシン命令、同図（ｃ）及び（ｄ）はローカルスタックの様子をそれぞれ示す図、第５図は第２図のステップＳ７の処理例を示す図で、同図（ａ）はプログラム例、同図（ｂ）は同プログラムに対応するマシン命令、同図（ｃ）はローカルスタックの様子をそれぞれ示す図、第６図は第２図のステップＳ６でＣＰが無い場合の処理例を示す図で、同図（ａ）はプログラム例、同図（ｂ）は同プログラムに対応するマシン命令、同図（ｃ）はローカルスタックの様子をそれぞれ示す図、第７図及び第８図はカットオペレータを含まない節を処理するためのマシン命令をそれぞれ示す図、第９図は環境フレームの構成図、第１０図は選択点フレームの構成図、第１１図は従来のＰＲＯＬＯＧ処理装置でプログラムをマシン命令に展開した図、第１２図はプログラム例とローカルスタックの様子とを示す図、第１３図はカットオペレータを含むプログラム例とローカルスタックの様子とを示す図である。

１…主メモリ、２…命令デコード部、３…レジスタファイル、４…ＡＬＵ、５…データバス。

出願人代理人　弁理士　鈴江武彦



第　1　図



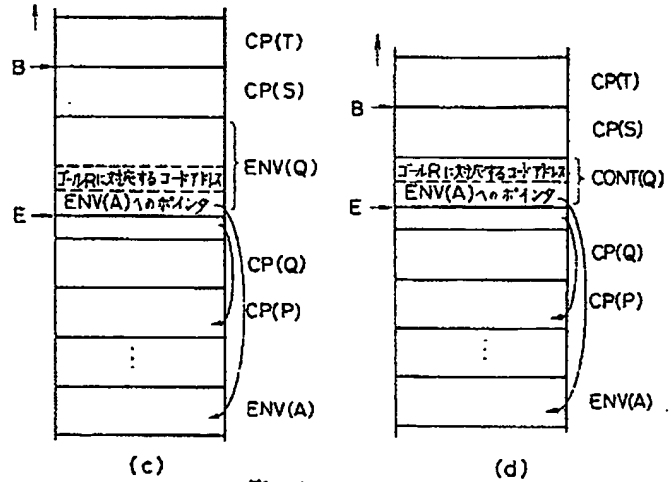
第 2 図

```
1  A :- ~ P. O. R.~.
2  P.
3  P.
4  Q :- S. T. !. U.
5  Q.
6  S.
7  S.
8  T.
9  T.
10 U.
```

(a)

```
Q/n : try_me_else  L          (1)
      mark_prv_cp             (2)
      allocate                (3)
      get_args_of  Q          (4)
      put_args_of  S          (5)
      call         S          (6)
      put_args_of  T          (7)
      call         T          (8)
      cut                     (9)
      put_args_of  U          (10)
      deallocate              (11)
      execute      U          (12)
L:    trust_me_else fail
      get_args_of  Q
      proceed
```

(b)



(c) (d)

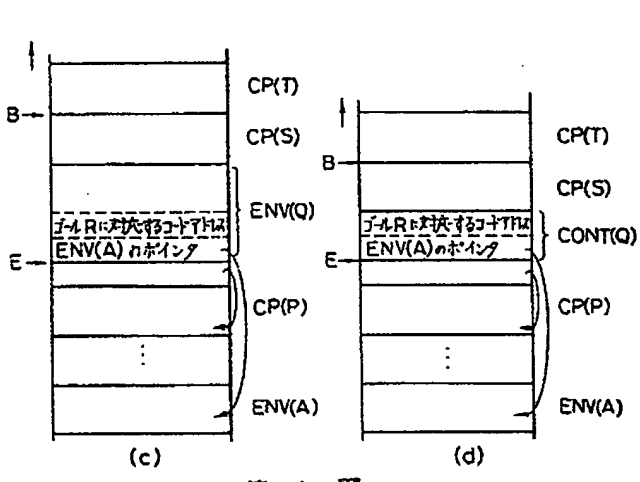
第 3 図

```
1  A :- ~ P. Q. R.~
2  P.
3  P.
4  Q.
5  Q :- S. T. !. U.
6  S.
7  S.
8  T.
9  T.
10 U.
```

(a)

```
Q/n : try_me_else  L
      get_args_of  Q
      proceed
L:    trust_me_else fail      (1)
      mark_cur_cp             (2)
      allocate                (3)
      get_args_of  Q          (4)
      put_args_of  S          (5)
      call         S          (6)
      put_args_of  T          (7)
      call         T          (8)
      cut                     (9)
      put_args_of  U          (10)
      deallocate              (11)
      execute      U          (12)
```

(b)



(c) (d)

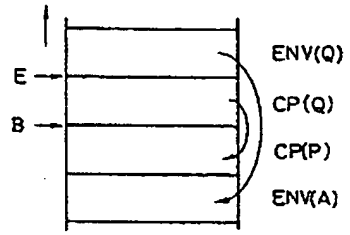
第 4 図

```
1 A:- ~ P, Q, R, ~.
2 P.
3 P.
4 Q:- var(X), !, X=Y.
5 Q.
6 R.
7 R.
```

(a)

```
Q/n : try_me_else                (1)
      allocate                   (2)
      get_args_of    Q           (3)
      put_args_of    var         (4)
      switch_on_term fail fail fail (5)
      dcut                       (6)
      deallocate                 (7)
      unify          X, Y        (8)
      proceed                    (9)
L:    trust_me_else  fail
      get_args_of    Q
      proceed
```

(b)



(c)

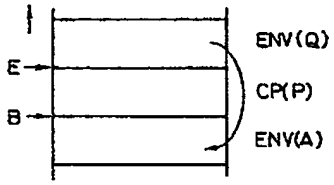
第 5 図

```
1 A:- ~ P, Q, R, ~.
2 P.
3 P.
4 Q.
5 Q:- var(X), !, X=Y.
6 R.
7 R.
```

(a)

```
Q/n : try_me_else   L
      get_args_of   Q
      proceed
L:    trust_me_else fail          (1)
      allocate                    (2)
      get_args_of   Q             (3)
      put_args_of   var           (4)
      switch_on_term fail fail fail (5)
      deallocate                  (6)
      unify         X, Y          (7)
      proceed                     (8)
```

(b)



(c)

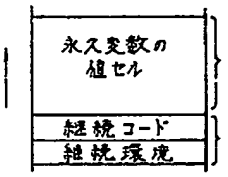
第 6 図

```
Q/n : try_me_else    L
      get_args_of    Q
      put_args_of    var
      switch_on_term fail fail fail
      dcut
      unify          X, Y
      proceed
```

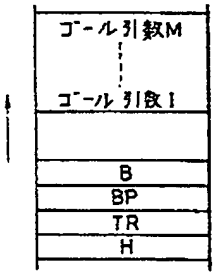
第 7 図

```
L:    trust_me_else fail
      get_args_of   Q
      put_args_of   var
      switch_on_term fail fail fail
      unify         X, Y
      proceed
```

第 8 図



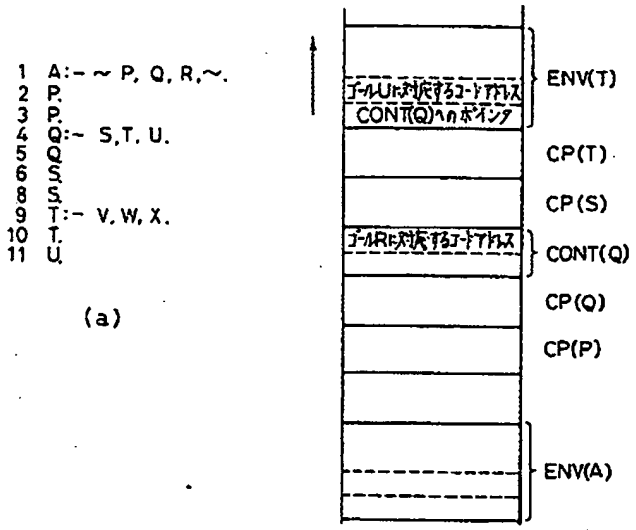
第 9 図



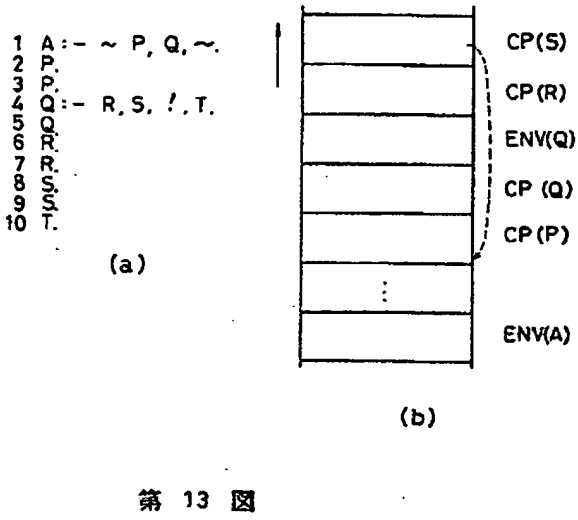
第 10 図

```
1  allocate
2  get_args_of   A
3  put_args_of   B
4  call          B
5  put_args_of   C
6  call          C
7  put_args_of   D
8  deallocate
9  execute       D
```

第 11 図


1 A:- ~ P, Q, R, ~.
2 P.
3 P.
4 Q:- S, T, U.
5 Q.
6 S.
8 S.
9 T:- V, W, X.
10 T.
11 U.
(a)
ENV(T)
CP(T)
CP(S)
CONT(Q)
CP(Q)
CP(P)
ENV(A)
(b)

第 12 図


1 A:- ~ P, Q, ~.
2 P.
3 P.
4 Q:- R, S, !, T.
5 Q.
6 R.
7 R.
8 S.
9 S.
10 T.
(a)
CP(S)
CP(R)
ENV(Q)
CP (Q)
CP(P)
ENV(A)
(b)

第 13 図